

Kapitel 7

Datengesteuerte Programmierung

Die konsequente Verwendung von Datenabstraktion isoliert die Benutzung von Datenwerten von Details ihrer konkreten Repräsentation. Typen helfen, unterschiedliche Sorten Daten zu unterscheiden und Verwechslungen zu vermeiden. Die Typabstraktionen allein sind — so wie sie sind — allerdings noch nicht mächtig genug für die Strukturierung komplizierter Systeme, in denen verschiedene Datenwerte verschiedener Typen auf die gleiche Art und Weise benutzt werden. Das einfachste Beispiel ist vielleicht die Verwendung von kartesischer oder polarer Repräsentation für komplexe Zahlen — beide Repräsentationen unterstützen die gleichen Operationen, erfordern aber unterschiedliche Realisierungen dieser Operationen. Die dadurch entstehende „Repräsentationsfrage“ wird mit zunehmender Komplexität eines Softwareprojekts immer schwieriger zu entscheiden. Ab einer bestimmten Größe kann es gar unmöglich werden, alle Repräsentationen im voraus festzulegen. Es sind also Abstraktionen erforderlich, die einen späteren Wechsel der Repräsentation ermöglichen, ohne daß dabei besonderer Aufwand entsteht oder gar Fehler eingeführt werden.

7.1 Repräsentationen für Mengen

In vielen Programmen werden als Datenstrukturen *Mengen* benötigt. Für Mengen bieten sich eine ganze Reihe verschiedener Repräsentationen an. Welche am besten geeignet ist, hängt in der Regel vom Verwendungszweck ab. Zwei Repräsentationen für Mengen sind bereits bekannt:

Listen Eine Liste kann die Menge ihrer Elemente repräsentieren.

Suchbäume Suchbäume — bekannt aus dem letzten Kapitel — erlauben oft einen effizienteren Elementtest als Listen.

Es gibt noch eine unübersehbare Menge weiterer Repräsentationen, zum Beispiel balancierte Suchbäume, Hash-Tabellen, Arrays etc. Zu den wichtigen Operationen gehören die Erzeugung der Repräsentation einer leeren Menge, das Hinzufügen von neuen Elementen und ein Elementtest.

Zu Demonstrationszwecken dient noch eine dritte, besonders einfache Repräsentation für eine Menge: ihre *charakteristische Funktion*. Die charakteristische Funktion einer Menge ist eine boolesche Funktion (also ein Prädikat), die auf Dinge des Universums angewendet, „wahr“ für alle Elemente der Menge und „falsch“ für alle anderen liefert. Als Scheme-Prozedur könnte eine charakteristische Funktion für die Menge $\{1, 5, 7\}$ im Universum der Zahlen folgendermaßen aussehen:

```
(define one-five-seven
  (lambda (x)
    (cond ((= x 1) #t)
          ((= x 5) #t)
          ((= x 7) #t)
          (else #f))))
```

Für die allgemeine Realisierung derart repräsentierter Mengen wird zunächst ein neuer Typ definiert:

```
(define cf-set-type (make-type 'cf-set))
```

Damit die charakteristische Funktion ein Element erkennen kann, braucht sie ein Gleichheitsprädikat für die Elemente des Universums, das zur charakteristischen Funktion dazugepackt wird:

```
(define make-cf-set
  (let ((construct (typed-value-maker cf-set-type)))
    (lambda (= cf)
      (construct (cons = cf)))))
```

Es fehlen noch Selektoren und ein Prädikat:

```
(define cf-set-value (typed-value-selector cf-set-type))
(define cf-set-cf
  (lambda (cf-set)
    (cdr (cf-set-value cf-set))))
(define cf-set-=
  (lambda (cf-set)
    (car (cf-set-value cf-set))))
```

```
(define cf-set? (typed-value-predicate cf-set-type))
```

Die leere Menge entsteht aus einer charakteristischen Funktion, die für jeden Wert des Universums #f liefert:

```
(define make-empty-cf-set
  (lambda (=)
    (make-cf-set
     = (lambda (x) #f))))
```

Der Elementtest extrahiert die charakteristische Funktion und wendet sie auf den zu testenden Wert an:

```
(define cf-set-member?
  (lambda (element cf-set)
    ((cf-set-cf cf-set) element)))
```

Zum Hinzufügen eines Elements zu einer Menge wird eine neue charakteristische Funktion erzeugt, die ggf. auf die charakteristische Funktion der alten Menge zurückgreift.

```
(define adjoin-cf-set
  (lambda (element cf-set)
    (let ((old-cf (cf-set-cf cf-set))
          (= (cf-set-= cf-set)))
      (make-cf-set
       =
       (lambda (x)
         (or (= x element)
              (old-cf element)))))))
```

Damit sind nun die Operationen für drei Repräsentationen von Mengen bekannt:

	leere Menge	Elementtest	Element hinzufügen
Listen	' ()	member	cons
Suchbäume	make-empty-search-tree	search-tree-member?	search-tree-insert
char. Funktionen	make-empty-cf-set	cf-set-member?	adjoin-cf-set

7.2 Repräsentationsunabhängigkeit

Bei der Verwendung von Mengen in Programmen ist häufig nicht sofort offensichtlich, welche Repräsentation am besten ist — ob nun Speichereffizienz, kurze Suchzeiten oder andere Eigenschaften für die Auswahl ausschlaggebend sind. Es wäre also gut, wenn das Programm sich noch nicht auf eine bestimmte Repräsentation festlegen könnte, so daß ein späterer Wechsel einfach möglich ist. Es ist sogar möglich, daß Mengen unterschiedlicher Repräsentation in einem Programm gleichzeitig vorkommen. Damit wird es nötig, über die verschiedenen Repräsentation zu abstrahieren.

Die Abstraktionen für Mengen sollten folgendermaßen aussehen:

```
(define adjoin-set
  (lambda (element set)
    ...))
```

```
(define set-member?
  (lambda (element set)
    ...))
```

Adjoin-set und set-member? sollen unabhängig von der Mengenrepräsentation funktionieren. Diese bei der Erzeugung einmal festgelegt, danach werden nur noch diese sogenannten *generischen* Prozeduren verwendet.

Die einfachste Methode, adjoin-set und set-member? zu realisieren, ist die Verwendung von Fallunterscheidungen zwischen den verschiedenen Repräsentationen. Voraussetzung die Existenz von Prädikaten für alle Repräsentationen¹:

```
(define adjoin-set
  (lambda (element set)
    ((cond
      ((cf-set? set) adjoin-cf-set)
      ((search-tree? set) search-tree-insert)
      ((list? set) cons))
     element set)))
```

```
(define set-member?
  (lambda (element set)
    ((cond
      ((cf-set? set) cf-set-member?)
      ((search-tree? set) search-tree-member?)
      ((list? set) member))
     element set)))
```

¹In diesem Fall zeigt sich ein konzeptuelles Problem bei der Repräsentation von getypten Werten, die unter Umständen mit Listen verwechselt werden können. Der Test auf die Listenrepräsentation ist deshalb ganz hinten. Damit funktionieren die Prozeduren zwar, aber es handelt sich um eine alles andere als ideale Lösung

Diese Art der Programmierung erlaubt die Konstruktion mächtiger Abstraktionen. Besonders elegant erscheint sie aber nicht aus mehreren Gründen:

- Jede dieser Prozeduren enthält letztlich den gleichen `cond`-Ausdruck.
- Die Erweiterung um weitere Repräsentationen erfordert Änderungen in allen beteiligten Prozeduren.

Das erste Problem läßt sich die Einführung einer Abstraktion wie der folgenden mildern, aber nicht wirklich lösen:

```
(define choose-set-op
  (lambda (set)
    (cf-set-op search-tree-op list-op)
    (cond
      ((cf-set? set) cf-set-op)
      ((search-tree? set) search-tree-op)
      ((list? set) list-op))))
```

Mit der Hilfe von `choose-set-op` lassen sich Operationen wie `set-member?` geringfügig weniger geschwätzig gestalten:

```
(define set-member?
  (lambda (element set)
    ((choose-set-op set)
     cf-set-member?
     search-tree-member?
     member)
    element set)))
```

Das zweite Problem bleibt bestehen. Die Ursache des Problems entspringt folgender Beobachtung: *Die Prozeduren bestimmen die Operationen, nicht die Daten.*

7.3 Datengesteuerte Programmierung und Message-Passing-Style

Das letzte im vorigen Abschnitt beschriebene Problem ist gravierend: Die dort verwendete Technik für die Realisierung generischer Prozeduren ist nicht *additiv*: Die Unterstützung neuer Repräsentationen erfordert die Veränderung alter Prozeduren. Dies ist in kleinen Programmen mit wenigen und relativ fixen Repräsentationsalternativen für die Daten noch kein Problem. Sobald aber die Vielfalt an Datentypen und Repräsentationen steigt, wird es ernst. Spätestens, wenn kein Programmierer mehr alle Repräsentationen kennt (eigentlich aus Abstraktionsgründen eine wünschenswerte Situation) versagt die Programmierertechnik von oben.

Glücklicherweise sind Prozeduren in Scheme auch Daten, es läßt sich also ein Rollentausch vornehmen:

```
(define list->list-set-proc
  (lambda (= list)
    (lambda (message)
      (cond
        ((equal? 'adjoin message)
         (lambda (element)
           (list->list-set-proc = (cons element list))))
        ((equal? 'member? message)
         (lambda (element)
```

```

      (member? = element list))))))
(define member?
  (lambda (= element list)
    (any (lambda (x)
          (= element x))
        list)))

```

```

(define make-empty-list-set-proc
  (lambda (=)
    (list->list-set-proc = '())))

```

Diese Mengen benutzen immer noch eine Liste für die Repräsentation, verstecken diese aber in einer Prozedur. Diese Prozedur akzeptiert einen einzelnen Parameter, eine *Nachricht*, welche eine Operation auswählt und diese als Prozedur zurückliefert. Dieser Programmierstil nennt sich *Message-Passing Style*. Die *adjoin-* und *member?*-Prozeduren schicken jeweils eine Nachricht, um die passende Operation aus einer solchen Menge zu extrahieren, und rufen diese dann auf:

```

(define adjoin-list-set-proc
  (lambda (element list-set-proc)
    ((list-set-proc 'adjoin) element)))

```

```

(define list-set-proc-member?
  (lambda (element list-set-proc)
    ((list-set-proc 'member?) element)))

```

Interessanterweise werden bei der Realisierung dieser Prozeduren gerade Operator und Operand vertauscht.

Das Prinzip läßt sich andere Repräsentationen übertragen. Eine weitere Repräsentation für Mengen basiert auf der charakteristischen Funktion (in Gestalt eines Prädikats) der Menge:

```

(define make-cf-set-proc
  (lambda (= cf)
    (lambda (message)
      (cond
        ((equal? 'adjoin message)
         (lambda (element)
           (make-cf-set-proc =
            (lambda (x)
              (or (= element x)
                  (cf element))))))
        ((equal? 'member? message)
         (lambda (element)
           (cf element)))))))

```

```

(define make-empty-cf-set-proc
  (lambda (=)
    (make-cf-set-proc = (lambda (x) #f))))

```

Die einzelnen Operationen sehen nun so aus:

```

(define adjoin-cf-set-proc
  (lambda (element cf-set-proc)
    ((cf-set-proc 'adjoin) element)))

```

```
(define cf-set-proc-member?  
  (lambda (element cf-set-proc)  
    ((cf-set-proc 'member?) element)))
```

Aber die Definitionen dieser Prozeduren sind *identisch* mit denen von `list` weiter oben! Sind also alle Repräsentationen von Mengen durch derartige Prozeduren realisiert, die Nachrichten `adjoin`, und `member?` akzeptieren, so läßt sich einfach schreiben:

```
(define adjoin-set  
  (lambda (element set)  
    ((set 'adjoin) element)))
```

```
(define set-member?  
  (lambda (element set)  
    ((set 'member?) element)))
```

Damit sind die Prozeduren wieder generisch. Sie verhalten sich je nach Repräsentation unterschiedlich. Wichtiger noch: sie müssen bei der Einführung neuer Repräsentationen nicht verändert werden, sind also *additiv*. Dieses allgemeine Prinzip — additive Repräsentationsunterscheidung — heißt *datengesteuerte Programmierung*.